A Linear Time Algorithm for Computing Off-line Speed Schedules Minimizing Energy Consumption

Bruno Gaujal, Alain Girault, Stéphan Plassart

A Linear Time Algorithm for Computing Off-line Speed Schedules Minimizing Energy Consumption*

Bruno Gaujal[1], Alain Girault[2], and Stéphan Plassart[3]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France.
bruno.gaujal@inria.fr
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France.
alain.girault@inria.fr
[3] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France.
stephan.plassart@inria.fr

Abstract

We consider the classical problem of minimizing off-line the total energy consumption required to execute a set of n real-time jobs on a single processor with varying speed. Each real-time job is defined by its release time, size, and deadline (all integers). The goal is to find a sequence of processor speeds, chosen among a finite set of available speeds, such that no job misses its deadline and the energy consumption is minimal. Such a sequence is called an optimal speed schedule. We propose a *linear time* algorithm that checks the schedulability of the given set of n jobs and computes an optimal speed schedule. The time complexity of our algorithm is in $\mathcal{O}(n)$, to be compared with $\mathcal{O}(n \log(n))$ for the best known solutions. Besides the complexity gain, the main interest of our algorithm is that it is based on a completely different idea: instead of computing the *critical intervals*, it sweeps the set of jobs and uses a *dynamic programming* approach to compute an optimal speed schedule. Our linear time algorithm is still valid (with some changes) with an arbitrary power function (not necessarily convex) and arbitrary switching times.

1 Introduction

Among numerous hardware and software techniques used to reduce energy consumption of a processor, supply voltage reduction, and hence reduction of CPU speed, is particularly effective. This is because the energy consumption of the processor is a function at least quadratic in the speed of the processor in most models of CMOS circuits. Nowadays, variable voltage processors are readily available and a lot of research has been conducted in the field of Dynamic Voltage and Frequency Scaling (DVFS). Under real-time constraints, the extent to which the system can reduce the CPU frequency (or speed in the following) depends on the jobs' features (execution time, arrival date, deadline) and on the underlying scheduling policy.

The problem of computing off-line DVFS schedules to minimize the energy consumption has been well studied in the literature, starting from the seminal paper of Yao et al. [1]. All the previous algorithms proposed in the literature compute the *critical interval* of the set of jobs[1], using more and more refined techniques to do so. This started in 1995 with [1] and [2] where it was independently shown that one can compute the optimal speed schedule with complexity $\mathcal{O}(n^3)$. Later, [3] showed in 2007 that the complexity can be reduced to $\mathcal{O}(n^2 L)$ (L being the

*This work has been partially supported by the LabEx PERSYVAL-Lab.

[1] The critical interval is the time interval with the highest load per time unit.

nesting level of the set of jobs). Finally the complexity has been reduced to $\mathcal{O}(n^2)$ in the most recent work in 2017 [4].

When the number of available speeds is *finite*, equal to m, [5] proposed in 2005 a $\mathcal{O}(mn \log n)$ algorithm. In their most recent work, the same authors showed in 2017 that the complexity can be further reduced to $\mathcal{O}(n \log(\max\{m, n\}))$ [4].

Here, we present a *dynamic programming* solution that sweeps the set of tasks in reverse order of their release times (forward sweep will not have a linear complexity here) and computes the best speed at each time step while checking feasibility. The complexity is linear in the number of jobs, equal to Kn, where the constant K depends on the maximal speed and on a bound on the maximal relative deadlines of the jobs.

Finally, we show in Section 5.1 that the power of dynamic programming allows us to generalize this approach to the case where switching from one speed to another is not free, but instead takes some time ρ and may also have an energy cost, which is a more realistic model. We also show in Section 5.2 that our model allows for arbitrary power functions, not necessarily convex in the speed.

2 System Model

We consider a set of n jobs $\{J_i\}_{i=1..n}$ to be executed by a single core processor equipped with dynamic voltage and frequency scaling (DVFS). Each job J_i is defined by the triplet (τ_i, c_i, d_i), where τ_i is the *inter-arrival time* between J_i and J_{i-1} (with $\tau_1 = 1$ by convention), c_i is the *size* (also called its WCET), and d_i the *relative deadline* bounded by Δ. From the inter-arrival times and the relative deadlines we can reconstruct the *release times* r_i and the *absolute deadlines* D_i of the job J_i: $r_i = \sum_{k=0}^{i} \tau_k \quad \forall i > 1$ and $D_i = r_i + d_i$. We also denote by T the last deadline among all jobs, called the *time horizon* of our system. It is defined by $T = \max_{i=1}^{n}\{D_i\}$. We assume that all these quantities are in $\mathbb{N}$.

The single core processor is equipped with m processing speeds also assumed to be in $\mathbb{N}$, and $s_{\max}$ denotes the maximal speed. The set of available speeds is denoted $\mathcal{S}$. The speeds are not necessarily consecutive integers. In the first part of the paper, we assume the cost of speed switching to be null. This will be generalized in Section 5 for a non-null speed switching thanks to the technique introduced in [6].

We use the *Earliest Deadline First* (EDF) preemptive scheduling policy. This approach as already been used in [6]. A key advantage of EDF is that it is optimal for *feasibility*. A set of jobs is *feasible* if and only if there exists a job schedule so that no deadline is missed when the processor always uses its maximal speed $s_{\max}$. In this respect, the optimality of EDF means that, if a set of jobs is feasible, then it is also feasible under EDF.

The power dissipated at any time t by the processor running at speed $s(t)$ is denoted $P_{ower}(s(t))$. For the time being, we assume that the P_{ower} function is *convex* (this assumption will be relaxed in Section 5.2). Under these notations, the total energy consumption E is:

$$E = \int_1^T P_{ower}(s(t))\mathrm{d}t. \tag{1}$$

Given a set of n jobs $\{J_i\}_{i=1..n}$, the goal is to find an *optimal speed schedule* $\{s^*(t), t \in [1, T]\}$ that will allow the processor to execute all the jobs before their deadlines while minimizing the total energy consumption E.

3 State Space

A natural idea is to store, at time t, the set of jobs present at time t, i.e., $\{J_i = (r_i, c_i, d_i)$, s.t. $r_i \leq t \leq r_i + d_i\}$. Yet, in order to compute the speed of the processor, one does not need to know the set of actual jobs but only the cumulative remaining *work* present at time t, corresponding to these jobs. Therefore, a more compact state will be the *remaining work function* $w_t(\cdot)$ at time t: for any $u \in \mathbb{R}^+$, $w_t(u)$ is the amount of work that must be executed before time $u + t$, taking into account all the jobs J_i present at time t (*i.e.*, with a release time $r_i \leq t$ and deadline $r_i + d_i > t$). By definition, the remaining work $w_t(\cdot)$ is a *staircase function*.

To derive a formula for $w_t(\cdot)$, let us introduce the work quantity that arrives at any time t: we define in Def. 1 a new function $a_t(\cdot)$. For any $u \in \mathbb{R}^+$, the quantity $a_t(u)$ is the amount of work that arrives at time t and must be executed before time $t + u$.

Definition 1. *The amount of work that arrived at time t and must be executed before time $t + u$ is*

$$a_t(u) = \sum_{i \,|\, r_i = t} c_i H_{d_i}(u), \tag{2}$$

where $H_{d_i}(\cdot)$ is the discontinuous step function defined $\forall u \in \mathbb{R}$ by

$$H_{d_i}(u) = \begin{cases} 0 & \text{if } u < d_i, \\ 1 & \text{if } u \geq d_i. \end{cases}$$

To illustrate the definition of $a_t(\cdot)$, let us consider an example with 3 jobs J_1, J_2, J_3 with respective release times $r_1 = r_2 = r_3 = t$, sizes $c_1 = 1$, $c_2 = 2$, $c_3 = 1$ and relative deadlines $d_1 = 2$, $d_2 = 3$, $d_3 = 5$. The corresponding function $a_t(\cdot)$ is displayed in the middle of Fig. 1.

Def. (1) allows us to describe the state change formula when moving from time $t - 1$ to time t, using speed $s(u)$ in the while interval $[t - 1, t]$.

Lemma 1. *At time $t \in \mathbb{N}$ the remaining work function is given by:*

$$w_t(\cdot) = \mathbb{T}\left[\left(w_{t-1}(\cdot) - \int_{t-1}^{t} s(u)du\right)^+\right] + a_t(\cdot), \tag{3}$$

with $\mathbb{T}f$ the shift on the time axis of function f, defined as: $\mathbb{T}f(t) = f(t+1)$ for all $t \in \mathbb{R}$, and $f^+ = \max(f, 0)$, the positive part of a function f.

Proof. Eq. (3) defines the evolution of the remaining work over time (see Fig. 1 for an illustration). The remaining work at time t is the remaining work at $t - 1$ minus the amount of work executed by the processor from $t - 1$ to t (which is exactly $\int_{t-1}^{t} s(u)du$) plus the work arriving at t. The "max" with 0 makes sure that the remaining work is always positive and the $\mathbb{T}$ operation performs a shift of the reference time from $t - 1$ to t. □

3.1 Size of the State Space

We denote by $\mathcal{W}$ the set of all possible remaining work functions that can be reached by any *feasible* set of jobs, when the processor only changes its speed at integer times, and when no job has missed its deadline before time t. The size of the state space $\mathcal{W}$ is denoted by Q.

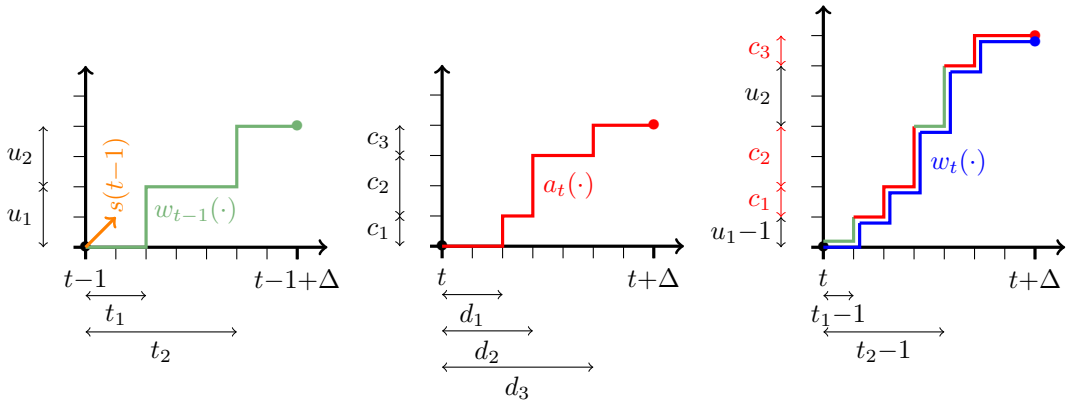

Figure 1: **Left**: State of the system at $t-1$. The green line depicts the remaining work function $w_{t-1}(\cdot)$. The constant speed chosen between times $t-1$ and t is $s(t-1)=1$; u_1 stands for $w_{t-1}(2)$ and u_2 stands for $w_{t-1}(5)-w_{t-1}(2)$. **Middle**: Arrival of three new jobs (r_i, c_i, d_i) at t: $J_1=(t,1,2)$, $J_2=(t,2,3)$, and $J_3=(t,1,5)$. The red line depicts the arrival work function $a_t(\cdot)$. **Right**: The blue line depicts the resulting state at t, $w_t(\cdot)$, obtained by shifting the time from $t-1$ to t, by executing 1 unit of work (because $s(t-1)=1$), and by incorporating the jobs arrived at t. Above the blue line are shown in green the "parts" of $w_t(\cdot)$ that come from $w_{t-1}(\cdot)$ and in red those from $a_t(\cdot)$.

Since all jobs have a relative deadline bounded by Δ, then for all $t \in \mathbb{N}$, and all $u \geq \Delta$, $w_t(u) = w_t(\Delta)$. Furthermore, w_t is a staircase function with steps at integer times,. Therefore, w_t is completely specified by its first values, $w_t(0), w_t(1), w_t(2), \ldots, w_t(\Delta)$.

If no job has missed its deadline before time t, then $w_t(0)=0$ (no work with deadline at most t is left at time t). Finally, the processor can execute at most $s_{\max}u$ amount of work during a time interval of size u. As a consequence, for feasible remaining work functions (*i.e.*, the remaining work functions corresponding to feasible jobs), one only needs to consider the remaining work functions that satisfy $\forall 0 < u \leq \Delta$, $w_t(u) \leq s_{\max}u$.

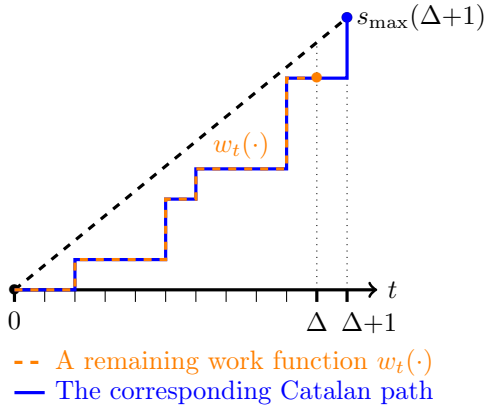

Figure 2: Bijection between remaining work functions $w_t(\cdot)$ (orange dashed staircase) and the Catalan paths (blue staircase).

Now, the set of feasible remaining work functions $w_t(\cdot)$ can be bijectively associated to the set of increasing paths over the 2D integer lattice[2] that start from point $(0,0)$, end into point $(\Delta+1, s_{\max}(\Delta+1))$, and that stay below the diagonal. This is done by adding one step to the $w_t(\cdot)$ staircase, from $(\Delta, w_t(\Delta))$ to $(\Delta+1, s_{\max}(\Delta+1))$. These paths are called *Catalan paths* in the following. This bijection is illustrated in Fig. 2, where the additional step connects the orange bullet and the blue bullet.

As a consequence, the size Q of the state space $\mathcal{W}$ can be computed using a generalization of the Catalan numbers [7]: The number of Catalan paths from $(0,0)$ to $(\Delta+1), s_{\max}(\Delta+1))$ (hence the number of all possible remaining work functions for any set of feasible jobs) is:

$$Q = \frac{1}{1+s_{\max}(\Delta+1)} \binom{(s_{\max}+1)(\Delta+1)}{\Delta+1}. \tag{4}$$

4 Dynamic Programming Solution

The goal of this section is to describe a dynamic program that computes an optimal speed schedule $s^*(t), t \in [1,T]$, such that $s^*(t)$ minimizes the energy consumption among all schedules where the speed may only change at integer times (the speed is therefore a piece-wise constant function). We distinguish the case where the speeds form a consecutive set (that is, $\mathcal{S} = \{0,1,\ldots,m-1\}$) and the case where they do not.

4.1 Consecutive Speeds

Algorithm 1 computes the optimal speed schedule. Before presenting its pseudocode, let us provide an informal description of the behavior of the system. Under a given piece-wise constant speed schedule $s(1), s(2), \ldots, s(T-1)$, the state of the system evolves as follows:

At time 0, no jobs are present in the system so the initial state function w_0 is the null function, which we represent by the null vector of size Δ: $w_0 = (0,\ldots,0)$ (see line 4).
The first job J_1 is released at time 1, maybe simultaneously with other jobs, so the new state function becomes $w_1 = w_0 + a_1$ according to Eq. (3). The case where several jobs are released at time 1 is taken care by the sum operator in Eq. (2) used to compute a_1.

At time 1, the speed of the processor is set to $s(1)$. The processor uses this speed up to time 2, incurring an energy consumption equal to $P_{ower}(s(1))$.

At time 2, the state function becomes $w_2 = \mathbb{T}(w_1 - s(1))^+ + a_2$ according to Eqs. (3) and (2), and so on and so forth up to time $T-1$, resulting in the sequence of state functions $w_1, w_2, \ldots, w_{T-1}$.

Now, let us denote by $E_t^*(w)$ the minimal energy consumption from time t to time T, if the state at time t is w, and if the optimal speed schedule is $s^*(t), s^*(t+1), \ldots, s^*(T-1)$. Of course, this partial optimal schedule is not known. But let us assume (using a backward induction) that the optimal speed schedule is actually known *for all possible states* $w \in \mathcal{W}$ at

[2]Increasing paths over a 2D integer lattice are staircases.

time t. It then becomes possible to compute the optimal speed schedule for all possible states between time $t-1$ and T using the maximum principle:

$$E^*_{t-1}(w) = \min_{s\in\mathcal{S}} \left(P_{ower}(s) + E^*_t(\mathbb{T}(w-s)^+ + a_t)\right) \quad (5)$$

$$s^*(t-1)(w) = \arg\min_{s\in\mathcal{S}} \left(P_{ower}(s) + E^*_t(\mathbb{T}(w-s)^+ + a_t)\right), \quad (6)$$

where $s^*(t)(w)$ denotes the optimal speed at time t if the current state is w.

When time 0 is reached, we have computed the optimal speed schedule between 0 and T for all possible initial states. To obtain an optimal speed schedule for the sequence of states $w_1, \ldots, w_{T-1}$, we just have to return the speeds $s^*(1)(w_1), \ldots, s^*(T-1)(w_{T-1})$ (see line 28). Note that, because of the arg min operator in Eq. (6), the optimal schedule is not necessarily unique.

This is what Algorithm 1 below does. E^* is computed using the backward induction described previously, which is a special case of the finite horizon policy evaluation algorithm provided in [8] (p. 80).

Algorithm 1 Dynamic programming algorithm computing the optimal speed schedule.

1: **input:** $\{J_i = (\tau_i, c_i, d_i), i = 1..n\}$ % Set of jobs to schedule
2: $r_i = \sum_{k=0}^{i} \tau_k$
3: $T \leftarrow \max_i(r_i + d_i)$ % Time horizon
4: $w_0 \leftarrow (0, \ldots, 0)$
5: **for all** $w \in \mathcal{W}$ **do**
6: $E^*_T(w) \leftarrow 0$ % Initialization of the energy at the horizon
7: **end for**
8: $t \leftarrow T$ % Start at the horizon
9: **while** $t \geq 1$ **do**
10: **for all** $w \in \mathcal{W}$ **do**
11: $E^*_{t-1}(w) \leftarrow +\infty$
12: **for all** $s \in \mathcal{P}(w)$ **do**
13: $w' \leftarrow \mathbb{T}\left[(w-s)^+\right] + a_t$ % Computation of the next state
14: **if** $w' \notin \mathcal{W}$ **then**
15: $E^*_t(w') \leftarrow +\infty$ % The next state is unfeasible
16: **end if**
17: **if** $E^*_{t-1}(w) > P_{ower}(s) + E^*_t(w')$ **then**
18: $E^*_{t-1}(w) \leftarrow P_{ower}(s) + E^*_t(w')$ % Update the energy in state w at $t-1$
19: $s^*(t-1)(w) \leftarrow s$ % Update the optimal speed in state w at $t-1$
20: **end if**
21: **end for**
22: **end for**
23: $t \leftarrow t-1$ % Backward computation
24: **end while**
25: **if** $E^*_1(w_1) = +\infty$ **then**
26: **return "not feasible"**
27: **else**
28: **return** $\{s^*(t)(w_t)\}_{t=1\ldots T}$
29: **end if**

The cases where the set of jobs is unfeasible are taken into account by setting the energy function $E_t^*(w')$ to infinity if the state w' is unfeasible, that is, if $w' \notin \mathcal{W}$ (see line 15) since $\mathcal{W}$ is the set of feasible states by definition.

If $s(t)(w)$ is the speed that the processor has to use at time t in state w, then the deadline constraint on the jobs imposes that $s(t)(w)$ must be large enough to execute the remaining work at the next time step, and cannot exceed the total work present at time t. This means:

$$\forall t, \forall w, \quad w(\Delta) \geq s(t)(w) \geq w(1). \tag{7}$$

This set of *admissible* speeds in state w will be denoted by $\mathcal{P}(w)$ and formally defined as:

$$\mathcal{P}(w) = \big\{ s \in \mathcal{S} \text{ s.t. } w(\Delta) \geq s \geq w(1) \big\}. \tag{8}$$

Our first result is Theorem 1, which states that Algorithm 1 computes the optimal speed schedule.

Theorem 1. *Assume that the speeds form a consecutive set,* i.e., $\mathcal{S} = \{0, 1, \ldots, m-1\}$. *If the set of jobs is not feasible, then Algorithm 1 outputs "not feasible". Otherwise it outputs an optimal speed schedule that minimizes the total energy consumption.*

Proof. **Case A: The set of jobs is not feasible.** Then, at some time t, the state w_t will get out of the set of feasible states, for all possible choices of speeds. Hence its value $E_t^*(w_t)$ will be set to infinity (see line 15) and this will propagate back to time 1. In conclusion, $E_1^*(w_1)$ will be infinite and Algorithm 1 will return "not feasible" (see line 26).

Case B: The set of jobs is feasible. The proof proceeds in two stages. In the first stage we show that there exists an optimal solution where speed changes only occur at integer times. In the second stage, we show that Algorithm 1 finds an optimal speed selection among all solutions that only allow speed changes at integer times.

Case B – first stage. To prove that there exists an optimal solution where speed changes only occur at integer times, let us first present the algorithm that computes the optimal speed schedule described in [1]. The core principle of this algorithm is to compute the *critical interval* I^c, defined to be the time interval with the highest average amount of work (in general there can be several such intervals, in which case we pick anyone).

To formally define of the critical interval, we rely on the release time r_i of job J_i and on its absolute deadline D_i.We say that a job J_i *belongs* to an interval $I = [u, v]$, denoted $J_i \in [u, v]$, iff $r_i \geq u$ and $D_i \leq v$. Using this notation, the critical interval I^c is:

$$I^c = [u^c, v^c] = \arg\max_{I=[u,v]} \frac{\sum_{J_i \in I} c_i}{v - u}. \tag{9}$$

Let ℓ be the length of I^c: $\ell = v^c - u^c$; and let ω be the total amount of work in I^c: $\omega = \sum_{J_i \in I^c} c_i$. Since the power is a convex function of the speed, the optimal speed s^c over I^c is *constant* and equal to $s^c = \frac{\omega}{\ell}$.

When the set $\mathcal{S}$ of available speeds is finite, the optimal solution is inferred from the unconstrained case as follows: Pick the two neighboring available speeds s_1 and s_2 in $\mathcal{S}$ such that s^c belongs to $[s_1, s_2)$. As a consequence, s^c is equal to a linear combination of s_1 and s_2:

$$s^c = \alpha s_1 + (1 - \alpha) s_2, \text{ with } \alpha = \frac{s_2 - s^c}{s_2 - s_1}. \tag{10}$$

An optimal speed schedule is therefore obtained by selecting speed s_1, possibly over several sub-intervals, for a cumulative time equal to $\alpha\ell$, and speed s_2 the rest of the time, for a total time equal to $(1-\alpha)\ell$.

Then, the critical interval I^c is *collapsed*, meaning that the working interval $[1,T]$ becomes the union $[1,u^c]\cup[v^c,T]$, and all the jobs included in I^c are removed from the set of jobs. The new critical interval is constructed over the contracted interval and over the remaining jobs, and so on and so forth.

This ends the description of the optimal solution and we are now ready for the proof that there exists an optimal solution whose speed changes occur at integer times.

Since the release times and deadlines are integers, the critical interval I^c has integer bounds: $I^c=[u^c,u^c+\ell]$ with $u^c,\ell\in\mathbb{N}$. Since the sizes of the jobs are also integer, the total amount of work over I^c is also integer: $\omega\in\mathbb{N}$.

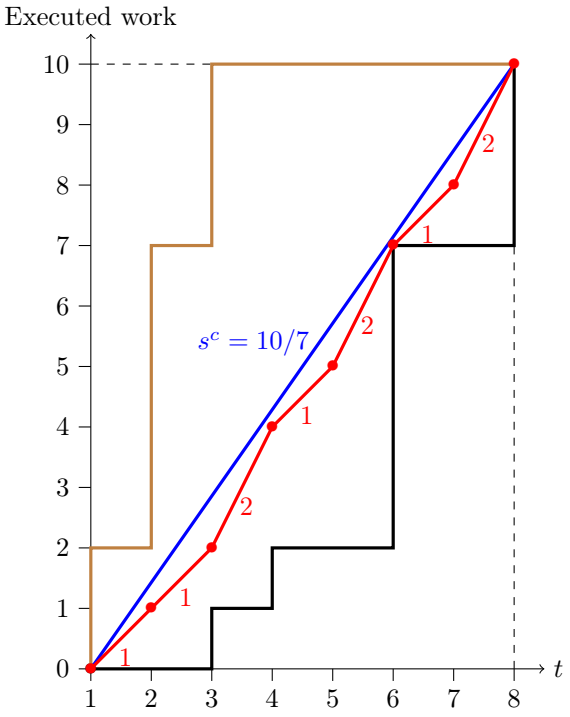

Figure 3: Construction of the optimal solution over a critical interval made of 4 jobs $\{J_i\}_{i=1..4}$ whose features $(r_i,c_i,d_i)_{i=1..4}$ are $(1,1,2)$, $(1,1,3)$, $(2,5,4)$, and $(3,3,5)$. The corresponding cumulative deadlines form the black staircase while the cumulative arrivals form the brown staircase. The critical interval is $I^c=[1,8]$, the total amount of work over I^c is $\omega=10$, the optimal speed is $s^c=10/7$, and its neighboring speeds are $s_1=1$ and $s_2=s_1+1=2$. The optimal speed schedule only uses speeds 1 and 2 and only changes speeds at integer times. The sequence of optimal speeds given by Eq. (12) is $(1,1,2,1,2,1,2)$.

When the set of available speeds is consecutive, $\mathcal{S}=\{0,1,\ldots,m-1\}$, the two neighboring

available speeds of $s^c = \frac{\omega}{\ell}$ satisfy $s_1 \leq s^c < s_2 = s_1 + 1$. In this case,

$$s^c = \frac{\omega}{\ell} = \alpha s_1 + (1-\alpha)(s_1+1) = s_1 + 1 - \alpha. \tag{11}$$

This implies that $\alpha\ell = \ell(s_1+1) - \omega$. Since $\ell, \omega, s_1 \in \mathbb{N}$, then $\alpha\ell \in \mathbb{N}$. This means that the speeds s_1 and s_2 will both be used during an integer amount of time. One optimal speed schedule can be constructed by using speed s_1 over $\alpha\ell$ intervals of length one, and speed s_1+1 over $(1-\alpha)\ell$ intervals of size one, constructed in the following manner: The speed $s^*(k) \in \{s_1, s_1+1\}$ used in interval $[u^c + k - 1, u^c + k]$, for $k = 1..\ell$, is:

$$s^*(k) = \lfloor k s^c \rfloor - \lfloor (k-1) s^c \rfloor. \tag{12}$$

This choice of speeds makes sure that speed s_1 is used during $\alpha\ell$ unit intervals and speed s_2 during $(1-\alpha)\ell$ unit intervals over the critical interval. In addition, under speeds $s^*(k)$, the jobs in the critical interval are all executed within their deadlines because of the following two reasons:

1. On the one hand, for any k strictly less than v^c, the sizes of the jobs in the critical interval with deadlines smaller than $k + u^c$ must sum up to a value V_k not larger than ks^c by non-criticality of the sub-interval $[u^c, u^c + k]$. Since the sizes of the jobs are integers, one further gets $V_k \leq \lfloor ks^c \rfloor$.

2. On the other hand, Eq. (12) implies that $s^*(1) + s^*(2) + \cdots + s^*(k) = \lfloor ks^c \rfloor$.

As a consequence, $V_k \leq s^*(1) + s^*(2) + \cdots + s^*(k)$, meaning that the sum of the sizes of the jobs belonging to the interval $[u^c, u^c + k]$ is less than the total amount of the work executed by the processor during the interval $[u^c, u^c + k]$.

Under this optimal solution, all the speed changes occur at integer points. The construction of this optimal solution is illustrated in Fig. 3: The integer cumulative deadlines (black staircase) are below the straight line whose slope is s^c (blue line) if and only they are also below the broken line with slopes $s^*(k)$ (red broken line).

Case B – second stage. In the second stage of the proof, we show that Algorithm 1 finds an optimal speed selection among all solutions that only allow speed changes at integer times. Together with the first stage, this will end the proof. Proving the optimality of Algorithm 1 is classical in dynamic programming. This is done by a backward induction on the time t. Let us show that $E_t^*(w)$, as computed by the algorithm, is the optimal energy consumption from time t to time T under any possible state w at time t.

Initial step: $t = T$. We set $E_T^*(w) = 0$ for all w. Indeed no jobs are present after time T, so that the state reached at time t must be $w_T = (0, 0, \ldots, 0)$ because no work is left at time T and the value $E_T^*(w_T) = 0$ is therefore correct. No speed has to be chosen at time T.

Induction: Assume that the property is true at time $t+1$. At time t, Algorithm 1 computes $\forall w \in \mathcal{W}$, $E_t^*(w)$. In particular, if the set of jobs is feasible, then the actual state at time t, w_t, must be in $\mathcal{W}$. Therefore, according to lines 17 and 18, we have:

$$E_t^*(w_t) = \min_{s \in \mathcal{P}(w)} \Big(P_{ower}(s) + E_{t+1}^*(\mathbb{T}(w_t - s)^+ + a_t) \Big).$$

All possible speeds at time t are tested with their optimal continuation (by induction hypothesis). Therefore, the best choice of speed at t, which minimizes the total energy from t to T, is selected by Algorithm 1.

Finally, when all the speed changes occur at integer times, the total energy consumption computed by Eq. (1) is equal to the value $E_1^*(w_1)$ computed by Algorithm 1. □

Theorem 2. *The time complexity of Algorithm 1 is Kn, where n is the number of jobs and the constant K depends on the maximal speed $s_{\max}$ and the maximal relative deadline Δ.*

Proof. The proof proceeds by inspecting Algorithm 1 line by line. The number of operations in line 13 is equal to the number of jobs whose release time is at time t, denoted $n_t = \#\{i | r_i = t\}$. The sum of all n_t is equal to the total number of jobs, $\sum_{t=1}^{T} n_t = n$.

Furthermore, the number of operations in line 14 is Δ (to check if $w'(i) \leq i s_{\max}$ for $i = 1..\Delta$). Therefore the total number O of arithmetic operations is:

$$O = \sum_{t=1}^{T} \sum_{w \in \mathcal{W}} \sum_{s \in \mathcal{P}(w)} (n_t + \Delta + K'), \tag{13}$$

where K' is a constant.

The size of $\mathcal{P}(w)$ is bounded by $s_{\max}$. Hence O is bounded by a linear function of n and T:

$$O \leq nQs_{\max} + TQs_{\max}(\Delta + K'). \tag{14}$$

We have seen previously that Q is bounded by a function of $s_{\max}$ and Δ (see Eq. (4)). Now, $T = \max_{i=1}^{n}(r_i + d_i) = \max_{i=1}^{n}(d_i + \sum_{j=1}^{i} \tau_j)$. If there exists j such that $\tau_j > \Delta$, then there exists an interval of time when the processor must be idle, between the end of the execution of the first $j-1$ jobs and the release time of the j^{th} job. In this case the problem can be split into two: all jobs from 1 to $j-1$ and all jobs from j to n.

This means that one can assume with no loss of generality that all inter-arrival times are smaller than Δ, hence $T \leq n\Delta$. In conclusion, the total number of arithmetic operations is bounded:

$$O \leq nK \quad \text{with } K = Qs_{\max}(\Delta^2 + \Delta K' + 1). \tag{15}$$

□

4.2 Non Consecutive Speeds

When the available speeds do not form a consecutive set, Algorithm 1 may not find an optimal schedule because it is possible that all the optimal speed schedules change speed at non integer times. Such solutions will not be found by Algorithm 1. In this section we show that it is possible to go back to the consecutive case by interpolating the power function. We assign to each non available speed a power consumption by using a linear interpolation. Let $s \in \mathbb{N}$ such that $s < s_{\max}$ and $s \notin \mathcal{S}$. Let $s_1, s_2 \in \mathcal{S}$ be the two neighboring available speeds such that $s_1 < s < s_2$. s is equal to a convex combination of s_1 and s_2: $s = \beta s_1 + (1-\beta)s_2$ with $\beta = \frac{s_2 - s}{s_2 - s_1}$. We set $P_{ower}(s) = \beta P_{ower}(s_1) + (1-\beta)P_{ower}(s_2)$.

Once this is done for each non available speed, we use Algorithm 1 to solve the problem with all consecutive speeds between 0 and $s_{\max}$, the unavailable speeds being seen as available with the power cost defined above.

Once the optimal speeds have been computed, the following transformation is done in each time step. In the time interval $[t, t+1)$, if the optimal speed $s^*(t)$ was not originally available, then it is replaced by its two neighboring available speeds s_1 and s_2 over time intervals $[t, t+\beta)$ and $[t+\beta, t+1)$ respectively. This is illustrated in Fig. 4. Since the deadlines are integers, no job will miss its deadline during the interval $(t, t+1)$.

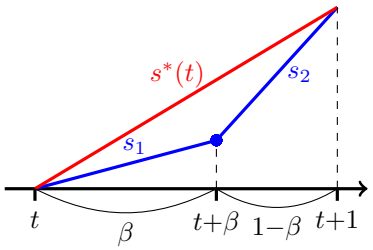

Figure 4: Amount of work executed with speed $s^*(t)$ (in red), and amount of work executed by the two neighboring available speeds s_1 and s_2 (in blue).

Theorem 3. *When the finite set $\mathcal{S}$ of available speeds is arbitrary, Algorithm 1 together with the interpolation transformation displayed in Fig. 4 computes an optimal speed schedule to execute n jobs in time $\mathcal{O}(n)$.*

Proof. Since the power cost $P_{ower}(s^*(t))$ is a linear interpolation of the power cost of the neighboring available speeds s_1 and s_2, the energy consumption over the interval $[t, t+1)$ is the same using speed s_t^* and using the neighboring speeds s_1 and s_2 over the two sub intervals $[t, t+\beta]$ and $[t+\beta, t+1]$. This means that the total energy consumption is the same before and after the transformation.

Theorem 1 states that, with consecutive speeds, the output of Algorithm 1 minimizes the total energy consumption. We have just shown that the transformation of each speed into a convex combination between two neighboring speeds provides a solution that only uses the available speeds and has the same total energy consumption as with consecutive speeds.

On the other hand, the optimal solution only using the subset composed by the available speeds must use at least as much energy as when all the intermediate speeds are available. This implies that Algorithm 1, used with the interpolated power function where unavailable speeds are replaced by their neighboring available speeds, gives an optimal solution that minimizes the total energy consumption.

Finally, this transformation takes a constant amount of time for each time interval $[t, t+1)$, therefore, the complexity remains linear in the number of jobs. □

5 Extensions

In this section, we show that Algorithm 1 can be adapted when switching from one speed to another has a time and/or energy cost, and when the power function is not convex.

5.1 Switching Time

So far, we have assumed that the time needed by the processor to change speeds is null. However, in all synchronous CMOS circuits, changing speeds does consume time and energy. The energy cost comes from the voltage regulator when switching voltage, while the time cost comes from the relocking of the Phase-Locked Loop when switching the frequency [9]. Burd and Brodersen have provided in [10] the equations to compute these two costs. In contrast with many DVFS studies (e.g., [10, 11, 12, 13]), our formulation can accommodate arbitrary energy cost to switch from speed s to s'. In the sequel, we denote this energy cost by $h_e(s, s')$.

As for the time cost, we denote by ρ the time needed by the processor to change speeds. For the sake of simplicity we assume that the delay ρ is the same for each pair of speeds, but our formalization can accommodate different values of ρ, as computed in [10].

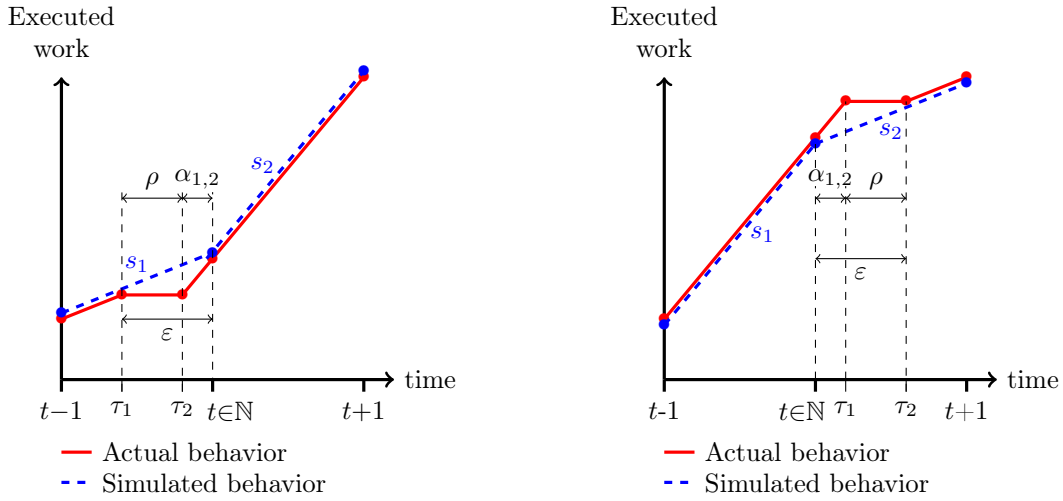

Figure 5: Transformation of the time delay into an energy additional cost by shifting the switching point. The left figure corresponds to the $s_1 < s_2$ case and the right figure to the $s_1 > s_2$ case. The red line represents the actual behavior of the processor with a ρ time delay. The blue dashed line represents an equivalent behavior in terms of executed work, with no time delay.

When there is a time delay, the executed work by the processor has *two* slope changes, at times τ_1 and τ_2, with $\tau_2 - \tau_1 = \rho$ (the red solid line in Fig. 5). From now on, we will only consider the case where speeds are consecutive. To take into account switching times inside integer intervals, as in § 4.2, we only have to modify the P_{ower} function with a penalty cost. We will come back to this case in the end of the section to give a precise expression for this penalty cost.

Since $\rho \notin \mathbb{N}$, we cannot have both $\tau_1 \in \mathbb{N}$ and $\tau_2 \in \mathbb{N}$. As a consequence, one of the remaining work functions $w_{\tau_1}(.)$ or $w_{\tau_2}(.)$ will not be integer valued. This is not allowed by our approach. We propose a solution inspired from [6] and illustrated in Fig. 5. It consists in *shifting* the time τ_1 when the speed change is initiated so that the global behavior can be simulated by a single speed change that occurs at an integer time (t in Fig. 5). The *actual* behavior of the processor is represented by the red solid line, while the *simulated* behavior, which is equivalent in terms

of the amount of work performed, is represented by the blue dashed line. The total amount of work done by the processor is identical in both cases at all integer times.

When $s_1 < s_2$, the speed change must be anticipated and occurs at $\tau_1 < t$ (left figure). When $s_1 > s_2$, the speed change has to be delayed and occurs at $\tau_1 > t$ (right figure). The exact computation of t_1 is similar in both cases and is straightforward (see [6]).

One issue remains however, due to the fact that the consumed energy will not be identical between the real behavior and the simulated behavior. Indeed, it will be higher for the actual behavior for convexity reasons. This additional energy cost of the real processor behavior must therefore be added to the energy cost of the equivalent simulated behavior. The value of ε, $\alpha_{1,2}$ are defined as in Fig. 5. In the case $s_2 > s_1$, $s_1\varepsilon = s_2\alpha_{1,2} = s_2(\varepsilon-\rho)$ so that $\varepsilon = \rho+\alpha_{1,2} = \frac{\rho s_2}{s_2-s_1}$. During the time delay ρ, the energy is consumed by the processor as if the speed were s_1. The additional energy cost incurred in the actual behavior (red curve) compared with the simulated behavior (blue curve), denoted $h_\rho(s_1, s_2)$, is therefore $h_\rho(s_1, s_2) = \alpha_{1,2}(P_{ower}(s_2) - P_{ower}(s_1))$. Using the value of $\alpha_{1,2}$, this yields $h_\rho(s_1, s_2) = \rho s_1 \left(\frac{P_{ower}(s_2)-P_{ower}(s_1)}{s_2-s_1}\right)$.

When $s_1 > s_2$, the additional cost becomes $h_\rho(s_1, s_2) = \rho s_2 \left(\frac{P_{ower}(s_1)-P_{ower}(s_2)}{s_1-s_2}\right)$.

This additional energy due to speed changes can be taken into account in our model in the cost function by modifying the state space $\mathcal{W}$: the new state at time t becomes the *pair* (w_t, s_{t-1}): the remaining work at time t (*i.e.*, w_t) and the previous speed that was used between $t-1$ to t (*i.e.*, s_{t-1}). Taking into account both the energy cost and the time cost, the new value computed at time t in Algorithm 1 becomes:

$$E^*_{t-1}(w, s) = \min_{s' \in \mathcal{P}(w)} \left(P_{ower}(s') + h_\rho(s, s') + h_e(s, s') + E^*_t(\mathbb{T}(w - s')^+ + a_t, s') \right), \qquad (16)$$

with $h_\rho(s, s') = 0$ when $s = s'$, and otherwise given above. The rest of the algorithm is unchanged and the complexity remains linear in the number of jobs. However, the constant factor K must be multiplied by m, since the new state space now contains all the pairs (x, s).

Finally, if the speeds are not consecutive, we show in Section 4.2 that speed changes can also happen inside an integer interval $[t, t+1]$ to mimic a non-available speed s by its two neighboring available speeds s_1, s_2 and using interpolation. Taking switching costs into account here is easier (no time shift is needed). One only needs to modify the definition of $P_{ower}(s)$ by adding the switching costs:

$$P_{ower}(s) = \alpha P_{ower}(s_1) + (1-\alpha)P_{ower}(s_2) + h_\rho(s_1, s_2) + h_e(s_1, s_2). \qquad (17)$$

5.2 Non-Convex Power Function

In most real processors, measurements of the power function show that it is not a convex function of the speed. In most cases, a more realistic approximation is $P_{ower}(s) = P_{stat}(s) + P_{dyn}(s)$, where $P_{dyn}(s)$ is convex but the leakage power $P_{stat}(s)$ depends on s and is not convex. If the power function is not convex, then it is well known that replacing $P_{ower}(\cdot)$ by its convex hull $\widehat{P_{ower}}(\cdot)$ (see for example [6]) and solving the speed selection problem with $\widehat{P_{ower}}(\cdot)$ instead of $P_{ower}(\cdot)$ also provides the optimal solution. This trick works as long as switching between speeds is free. As soon as switching has a cost (in energy or in time), one must be able to deal with non-convex power directly. Here, the assumption that the power is convex is not needed in our dynamic programming approach. In other words, Algorithm 1 and Theorem 3 are valid even when the P_{ower} function is arbitrary.

6 Conclusion

In this paper we have shown that the complexity of computing the best speed schedule to execute n jobs before their deadline while minimizing the total energy consumption can be done in time Kn. This result holds with an arbitrary power function and may also take into account speed switching costs. Although this result is satisfying because it achieves the theoretical lower bound, in practice, it suffers from a constant term K that is exponential in the maximal deadline of the jobs. This may hinder its applicability when the deadlines of the jobs are large. However this algorithm can be tuned to accommodate large deadlines by performing a state discretization that may result in a suboptimal result: the finer the discretization, the better the performance.

References

[1] F. F. Yao, A. J. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *36th Annual Symposium on Foundations of Computer Science*, (Milwaukee (WI), USA), pp. 374–382, IEEE Computer Society, Oct. 1995.

[2] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publisher, 1998.

[3]

[4] M. Li, F. F. Yao, and H. Yuan, "An $O(n^2)$ algorithm for computing optimal continuous voltage schedules," in *Annual Conference on Theory and Applications of Models of Computation, TAMC'17*, vol. 10185 of *LNCS*, (Bern, Switzerland), pp. 389–400, Apr. 2017.

[5] M. Li and F. F. Yao, "An efficient algorithm for computing optimal discrete voltage schedules," *SIAM J. Comput.*, vol. 35, pp. 658–671, 2005.

[6] B. Gaujal, A. Girault, and S. Plassart, "Dynamic speed scaling minimizing expected energy consumption for real-time tasks," Tech. Rep. HAL-01615835, Inria, 2017.

[7] P. Hilton and J. Pedersen, "Catalan numbers, their generalization, and their uses," *The Mathematical Intelligencer*, vol. 13, pp. 64–75, Mar 1991.

[8] M. L. Puterman, *Markov Decision Process : Discrete Stochastic Dynamic Programming*. Wiley, series in probability and statistics ed., February 2005.

[9] Q. Wu, P. Juang, M. Martonosi, and D. Clark, "Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors," in *International Conference on High-Performance Computer Architecture, HPCA'05*, (San Francisco (CA), USA), pp. 178–189, IEEE, Feb. 2005.

[10] T. Burd and R. Brodersen, "Design issues for dynamic voltage scaling," in *International Symposium on Low Power Electronics and Design, ISLPED'00*, (Rapallo, Italy), July 2000.

[11] M. Bandari, R. Simon, and H. Aydin, "Energy management of embedded wireless systems through voltage and modulation scaling under probabilistic workloads," in *International Green Computing Conference, IGCC'14*, (Dallas (TX), USA), pp. 1–10, IEEE Computer Society, Nov. 2014.

[12] K. Li, "Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels," *J. of Parallel and Distributed Computing*, vol. 95, pp. 15–28, Sept. 2016.

[13] J. Wang, P. Roop, and A. Girault, "Energy and timing aware synchronous programming," in *International Conference on Embedded Software, EMSOFT'16*, (Pittsburgh (PA), USA), ACM, Oct. 2016.